CS2040$_S$ Midterm Cheatsheet

github.com/reidenong/cheatsheets, AY23/24 S2

Time Complexity

Big-O Definition:
$T(n) = O(f(n))$ if $\exists c, n_0 > 0$ s.t. $\forall n > n_0$,
$$T(n) \leq cf(n)$$

Big-Ω Definition:
$T(n) = \Omega(f(n))$ if $\exists c, n_0 > 0$ s.t. $\forall n > n_0$,
$$T(n) \geq cf(n)$$

Big-Θ Definition:
$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Order of Growth:

$O(1)$	Constant time
$O(\log \log N)$	Double log
$O(\log N)$	Logarithmic
$O(\log^2 N)$	Polylogarithmic
$O(\sqrt{N})$	
$O(N)$	Linear
$O(N \log N)$	Log - linear
$O(N^2)$	Polynomial
$O(2^N)$	Exponential time
$O(2^{2N})$	
$O(N!)$	Factorial time

Specific Big Os:
- $\sum_i^N \frac{1}{i} = O(\log N)$
- $T(n) = T(n-1) + T(n-2) + O(1) = O(2^n)$

Preconditions:
- Fact that is true when the function begins
- Must be true for the function to work correctly

Postconditions:
- Fact that is true when the function ends
- Something useful to show that the computation was done correctly

Invariants:
- Relationship between variables that is always true
- A loop invariant is a condition that is true before and after each iteration of a loop

Searching

$O(\log N)$ **Binary Search:**

```
int binarySearch(int[] arr, int x)
  int lo = 0, hi = arr.length - 1;
  while (lo < hi)
      int mid = lo + (hi - lo)/2;
      if (in lower half)
          end = mid;
      else
          start = mid + 1;
  return lo;
```

Kth smallest element:
- DnC, partition array into 2 halves, recurse on the half that contains the kth element
- $O(N)$

Peak Finding:
- Operates on same concept as binary Search, DnC
- $O(\log N)$

2D Peak Finding $n \times m$:
- Naive: $O(N \log M)$
- Quadrant Divide and Conquer: $O(N + M)$

Sorting

Bubble Sort
- Repeatedly steps through the list, compares each pair of adjacent items and swaps them if they are in the wrong order.
- Invariant: At the end of iteration j, the last j elements are in their correct position.

ie. Globally sorted suffix

Best case: $O(N)$
Worst case: $O(N^2)$
Space: In-place
Stable: Yes

Selection Sort
- Repeatedly finds the minimum element from the unsorted part and puts it at the beginning.
- Invariant: At the end of iteration j, the first j elements are in their correct position.

ie. Globally sorted prefix

Best case: $O(N^2)$
Worst case: $O(N^2)$
Space: In-place
Stable: No

Insertion Sort
- Push the latest item into the sorted prefix one element at a time.
- Invariant: At the end of iteration j, the first j elements are sorted locally.

ie. Locally sorted prefix
- Very fast on almost-sorted arrays

Best case: $O(N)$
Worst case: $O(N^2)$
Average case: $\sum_{j=2}^{N} \Theta\left(\frac{j}{2}\right) = \Theta(N^2)$
Space: In-place
Stable: Yes

Merge Sort
- Divide and conquer, splitting the array into halves then sorting each individual half before merger
- **Locally sorted prefixes in powers of two**

Best case: $O(N \log N)$
Worst case: $O(N \log N)$
Space: $O(N \log N)$
Stable: Yes
* Merge sort may perform slower for small $N (< 1024)$ due the need to cache performance, branch prediction, general overhead costs

QuickSort
- DnC, pick a pivot x and partition the array into $> x$ and $< x$ partitions with end to end swapping. Then recurse in both partitions.
- Invariant: for each i, $arr[i] \leq x$ for $i < p$ and $arr[i] > x$ for $i > p$

ie. Sorted around pivots, which are in position
- Optimizations: Randomized pivot, 3-way partitioning, insertion sort for small N

Best case: $O(N \log N)$
Worst case: $O(N^2)$
Stable: No

Counting Sort
- Count the number of occurrences of each element and then use the counts to compute the position of each element in the output array.

Time: $O(N + k)$ where k is the range of the input
Space: $O(N + k)$
Stable: Yes

Radix Sort
- Sort the input numbers by their individual digits.

Time: $O(Nd)$ where d is the number of digits
Space: $O(N + k)$
Stable: Yes

Trees and balancing

Binary Search Tree:
$O(h)$ / $O(N)$
- insert, delete
- predecessor, successor, search
- findMax, findMin

Strictly $O(N)$:
- Traversal

bBST / AVL Trees:

A BST is balanced if $h = O(\log N)$.

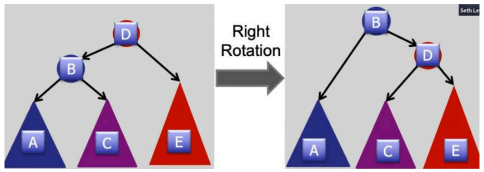
A node is height-balanced if the height of its left and right subtrees differ by at most 1. A tree is height-balanced if all its nodes are height-balanced.

A height balanced tree with N nodes has at most height $h < 2 \log N \leftrightarrow$ A height balanced tree with height h has at least $N > 2^{\frac{h}{2}}$ nodes.

Upper bound of nodes in a AVL tree:
$$N_h \leq 1 + 2N_{h-1} \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Lower bound of nodes in a AVL tree:
$$N_h \geq 1 + N_{h-1} + N_{h-2} \geq 2N_{h-2} = 2^{\frac{h}{2}}$$

Rotations




```
func rightRotate(D) :
    B = D.left
    B.parent = D.parent
    D.parent = b
    D.left = B.right
    B.right = D
    return B
func leftRotate(B) :
    D = B.right
    D.parent = B.parent
    B.parent = D
    B.right = D.left
    D.left = B
    return D
```

Balancing AVL Trees

WLOG, a node v is **left heavy** if left subtree has larger height than right subtree

If v is left heavy :
(1) v.left is balanced or left heavy : rightRotate(v)
(2) v.left is right heavy : leftRotate(v.left), rightRotate(v)

If v is right heavy :
(1) v.right is balanced or right heavy : leftRotate(v)
(2) v.right is left heavy : rightRotate(v.right), leftRotate(v)

Insertion:
- Insert node
- Walk up tree, only need to fix lowest unbalanced node
- Maxmimum 2 rotations

Deletion:
- Delete node
- Fix all unbalanced nodes until root
- Maximum $O(\log N)$ rotations

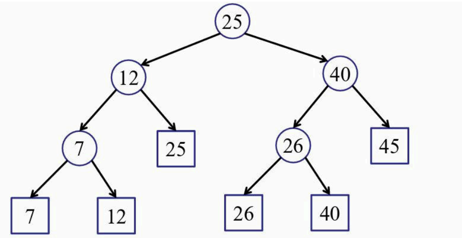
Trie:
- Independent of number of elements
- $O(L)$ where L is the length of the key
- Faster than the $O(Lh)$ alternative of strings in a bBST, though it has more nodes and thus more overhead space

Order statistics:
- select(i) : find the ith smallest element
- rank(x) : find the rank of element x
- $O(\log N)$ with bBST

Interval Queries:
- Sort all intervals by left endpoint in bBST
- For each node, store the maximum right endpoint in the subtree rooted at that node
- $O(\log N)$

(Dynamic) 1D Range Queries

Finding all elements in a range $[a, b]$.



- All elements are leaves. Each internal node stores the maximum value in its left subtree
- Step 1: Find split node ($O(\log N)$)
- Step 2: Do left and right traversals
- Invariant: Search interval for a left-traversal at node v includes the maximum item in the in the subtree rooted at v

Preprocessing:

- $O(N \log N)$ for N insertions of $O(\log N)$

Query:

- $O(\log N + k)$ where k is the number of elements in the range
- Tree can be augmented with the count of each node in subtree to support counting queries in $O(\log N)$

Space:

- $O(N)$

(Static) 2D Range Queries:

- Build a 1D x-tree for all x-coords
- For each internal node, build a y-tree for all y-coords

Preprocessing:

- $O(N \log N)$

Query:

- $O(\log N)$ to find split node
- $O(\log N)$ recursing steps
- $O(\log N)$ y-tree searches each of $O(\log N)$
- $O(k)$ enumerating output
- Total: $O(\log^2 N + k)$

Space:

- $O(N \log N)$

Modification:

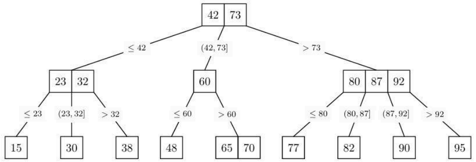
- $O(N)$ for rebuilding y-trees for the rotated nodes

D - dimension Range Queries:

In general for a $d -$ dimension range query,

- Query cost: $O(\log^d N + k)$
- Preprocessing: $O(N \log^{d-1} N)$
- Space: $O(N \log^{d-1} N)$

(a, b)-Trees



- A node can have at least a and at most b children, where $2 \le a \le \frac{b+1}{2}$.
- With sorted keys $v_1, v_2, ..., v_k$, v_1 has key range $\le v_1$, v_k has key range $> v_k$, and all other keys have range $(v_{i-1}, v_i]$
- All leaf nodes must be at the same depth

Operations

Search:

- $O(b \log_a N) = O(\log N)$ for N elements.

(Proactive) Insertion:

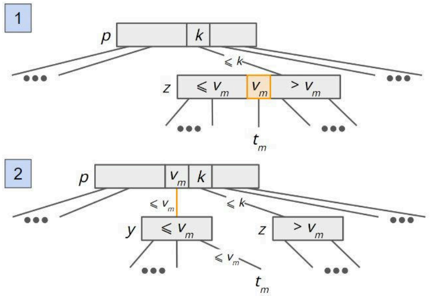
- Insertion may cause a node to have too many keys.
- We preemptively split full nodes (ie. $b - 1$ keys), guaranteeing parent of the node to be split will not have too many keys after insertion of split keys
- $O(\log N)$

```
### Insertion in a (a, b)-tree with root w
w = root
while true :
    if w contains b-1 keys :
        y, z = split(w)
        if x <= median :
            w = y
        else :
            w = z
    if w is a leaf :
        break
    else :
        w = getSubtree(w, x)
        # get the child node x should be in
w.insert(x)
```

Split:

- $O(b)$ for splitting a node with b children
- Occurs when a node u has b children and a new child is added
- We offload median node to parent, and split the remaining children into 2 nodes
- If we are splitting z, then preconditions are

1. z has $\ge 2a$ keys. After splitting and offering a key to parent, LHS has $\ge a - 1$ keys and RHS has $\ge a$ keys
2. z's parent has $\le b - 2$ keys.



(Lazy) Deletion:

- Deletion risks having nodes shrink too small.
- We use a passive strategy where we first delete target key, then recursively check upwards for violation while carrying out merge/share operations.
- To delete internal node, we replace with predecessor/ successor and delete leaf node
- $O(\log N)$

```
### Deleting key x in a (a, b)-tree
# Find node containing key x
w = search(x)
# Preprocessing for internal nodes
if w is internal :
    pre_node, pre_key = getPredecessor(w, x)
    swapkeys(w, x, pre_node, pre_key)
    w = pre_node
# Delete key
deleteKey(w, x)
# Fixing violations with merge/share
while true :
    if w contains < a-1 keys :
        z = getSmallestSibling(w)
        if w and z contain < b-1 keys :
            w = merge(w, z)
        else :
            w = share(w, z)
            # w can be either L or R node
    else :
        break
    # recurse upwards
    if w is not root :
        w = w.parent
    else :
        break
```

Merge / Share:

Suppose we are deleting a key from node z, which also has smallest sibling y. Deletion risks having nodes shrink to small.

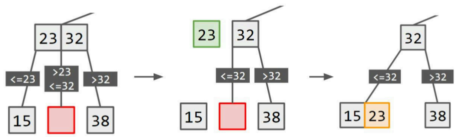
Given that z has $< a - 1$ keys,

Case (1) : z and y have $< b - 1$ keys together.

`merge(y, z)`

Algo:

1. In parent node of y and z, delete the key v that separates y and z
2. Add v to keylist of y
3. Add all keys of z to y
4. Delete z from parent node



Case (2) : z and y have $\ge b - 1$ keys together.

`share(y, z)`

Algo:

1. `merge(y, z)` gives us a node w with $\ge b$ keys
2. `split(w)` gives us nodes y and z with $\ge a - 1$ keys